Chap. 06

チーム開発とプロジェクト

中山浩太郎
松尾研究室

注意事項

UTokyo Wifi利用者 → 更新

プロジェクト申し込みフォーム

https://goo.gl/pRpYQq

（締め切り：11/06）

Gitを研究・開発で
日常的に使っている人

今日のコンテンツ

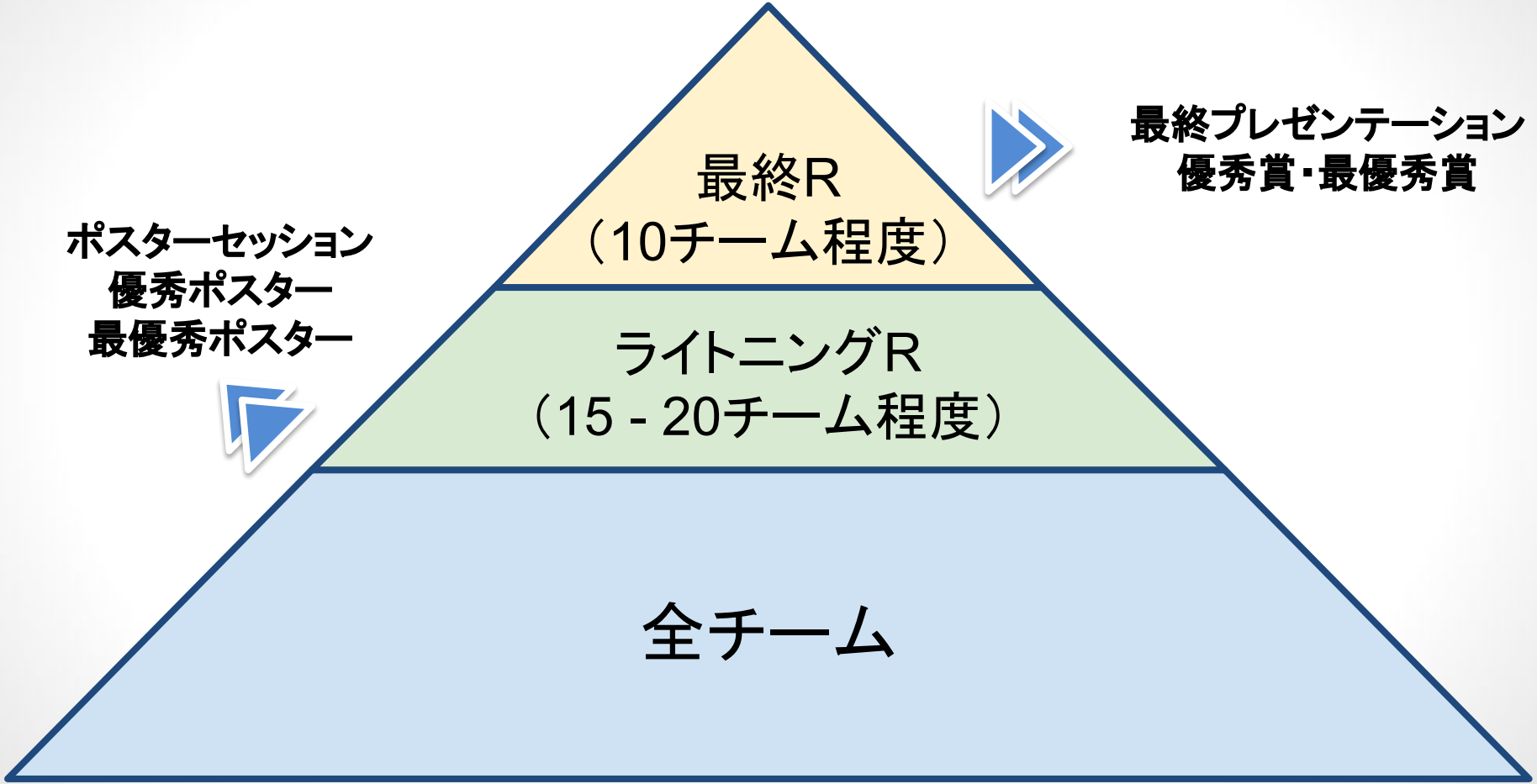
- プロジェクトについて
- Git入門
- Gitの作法
- アジャイル開発
- Deep Learningプロジェクト注意
- まとめ
- チームでの議論タイム

今日のコンテンツ

- プロジェクトについて
- Git入門
- Gitの作法
- アジャイル開発
- Deep Learningプロジェクト注意
- まとめ
- チームでの議論タイム

プロジェクトについて

- お題「Deep Learning技術を利用したアプリ」
- 評価項目
  - 新規性・有用性・技術的困難度
- 自分の強みを活かしてください
- プライベートチャンネルを利用しても良いです
  → 講師・TA追加
- Githubリポジトリはプライベートでも良いです
  → 講師・TA追加
- コミットのログを見て成績を判断します


最終R
（10チーム程度）
ライトニングR
（15 - 20チーム程度）
全チーム
最終プレゼンテーション
優秀賞・最優秀賞
ポスターセッション
優秀ポスター
最優秀ポスター

注意事項

プロジェクト申し込みフォーム

https://goo.gl/pRpYQq

（締め切り：11/06）

今日のコンテンツ

- プロジェクトについて
- Git入門
- Gitの作法
- アジャイル開発
- Deep Learningプロジェクト注意
- まとめ
- チームでの議論タイム

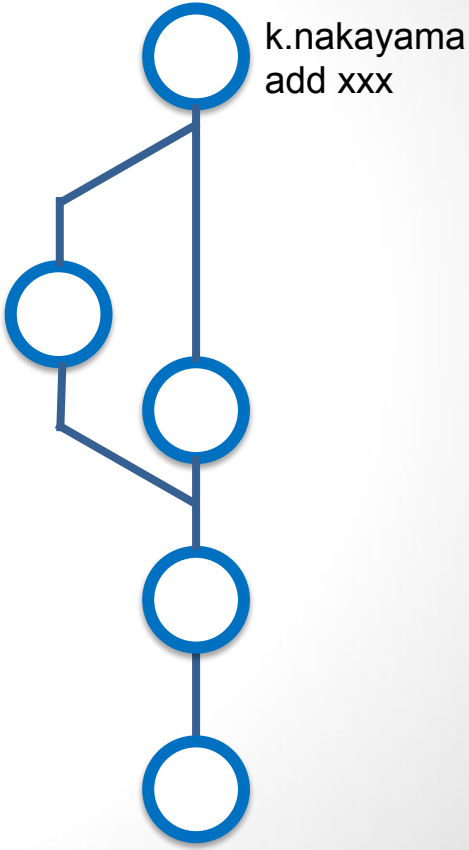
プロジェクトの問題

- どうやって開発を進めるか
  - コミュニケーションの方法
  - チームワーク
  - コードの共有方法
- 開発の規模と方向性
- 検証

段取り・進め方が大事

Gitとは

- 分散バージョン管理システム
- 大事な機能
  - ファイルのバージョン管理
  - いつでも前の状態に復元できる
  - バージョンの間で比較ができる
  - ブランチを分けることができる
    （開発の方向性を分けることができる）
  - ブランチを統合することができる
  - 誰がいつどの変更をしたかがわかる



Git（VCS）が無い世界

- 「これ変更したの誰？」→ 喧嘩
  → 実は自分だった → 恥ずかしい
- 「前のファイルも残しておこう」
  → run1.py, run2.py, run3.py, …→カオス
- 同時に一つのファイル編集 → 変更内容が全部消えた / 上書き → 悲しい
- 「最新のコードはどこ！？」

Gitと研究、バージョン管理

論文投稿
↓
数ヶ月後に査読がかえってくる
↓
既にコードは進んでいて実験再現不能
↓
論文投稿時の状態に戻せる、Gitならね

Gitと研究、バージョン管理

最近では論文原稿や本の執筆もGit

↓

添削もGit

↓

Diffを見れば変更内容が一目瞭然

Q1）Git以外でも良いのでは？

↓

分散でのチーム開発に必須

Q2) Github以外でも
良いのでは？
↓
エコシステムとしての完成度
リソースの関係上、Githubで

Gitの基本

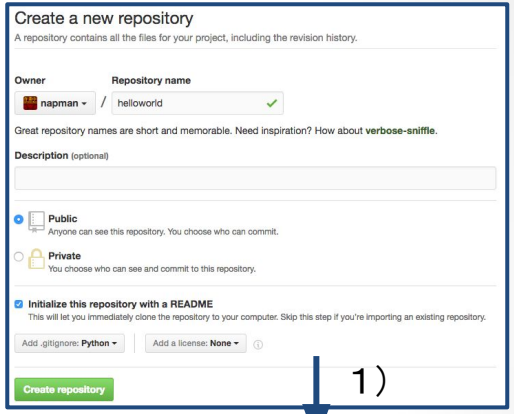
Add, Commit, Push

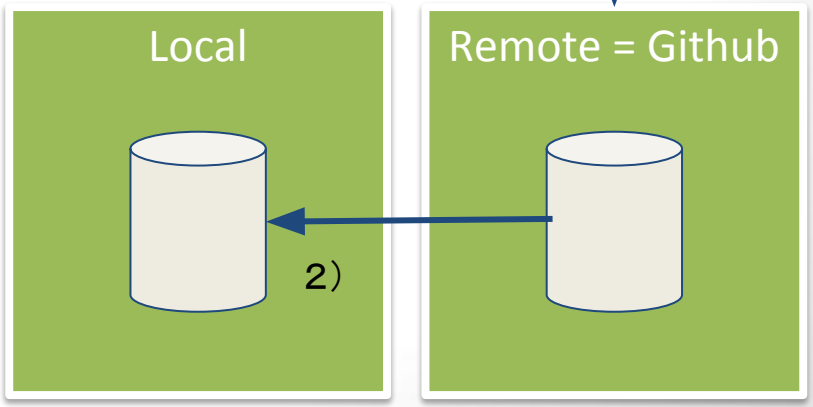
Gitの基本

1）Githubにリポジトリを作成

2）リポジトリをローカルにクローン

```
git clone https://github.com/napman/helloworld.git
```





Gitの基本

3）ファイル変更（追加・修正・削除・移動）

4）Add

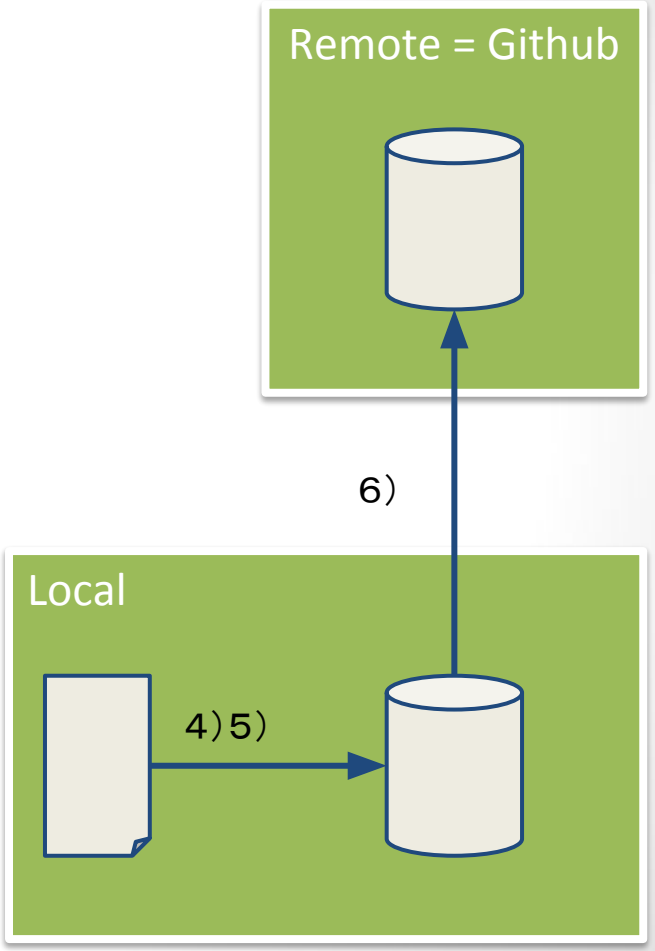
```
git add hello.py
```

5）Commit

```
git commit -m add hello.py
```

6）Push

```
git push
```



Exercise

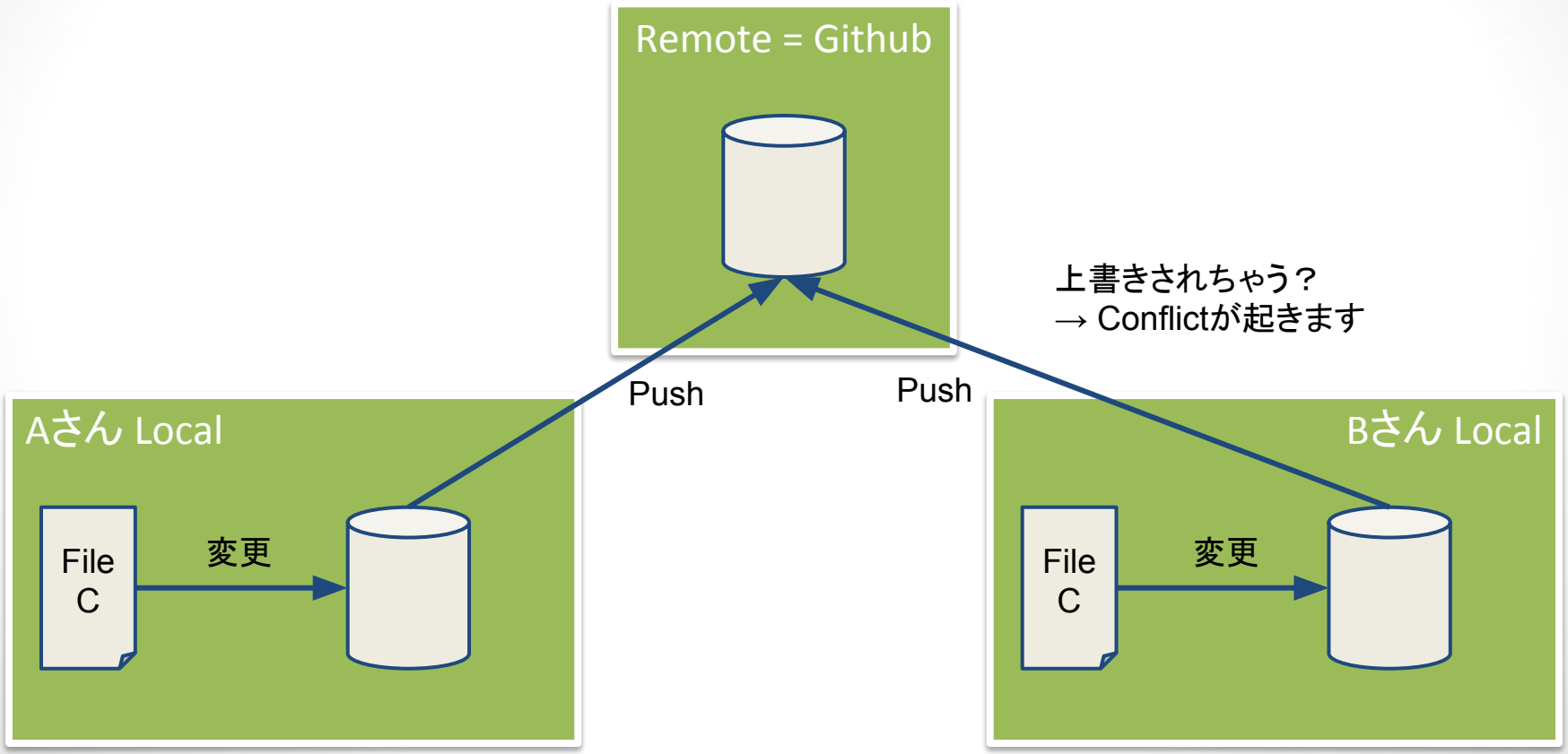
***Github*にリポジトリ作成**
***Add*、*Commit*、*Push*までやってみる**

Github Repository make

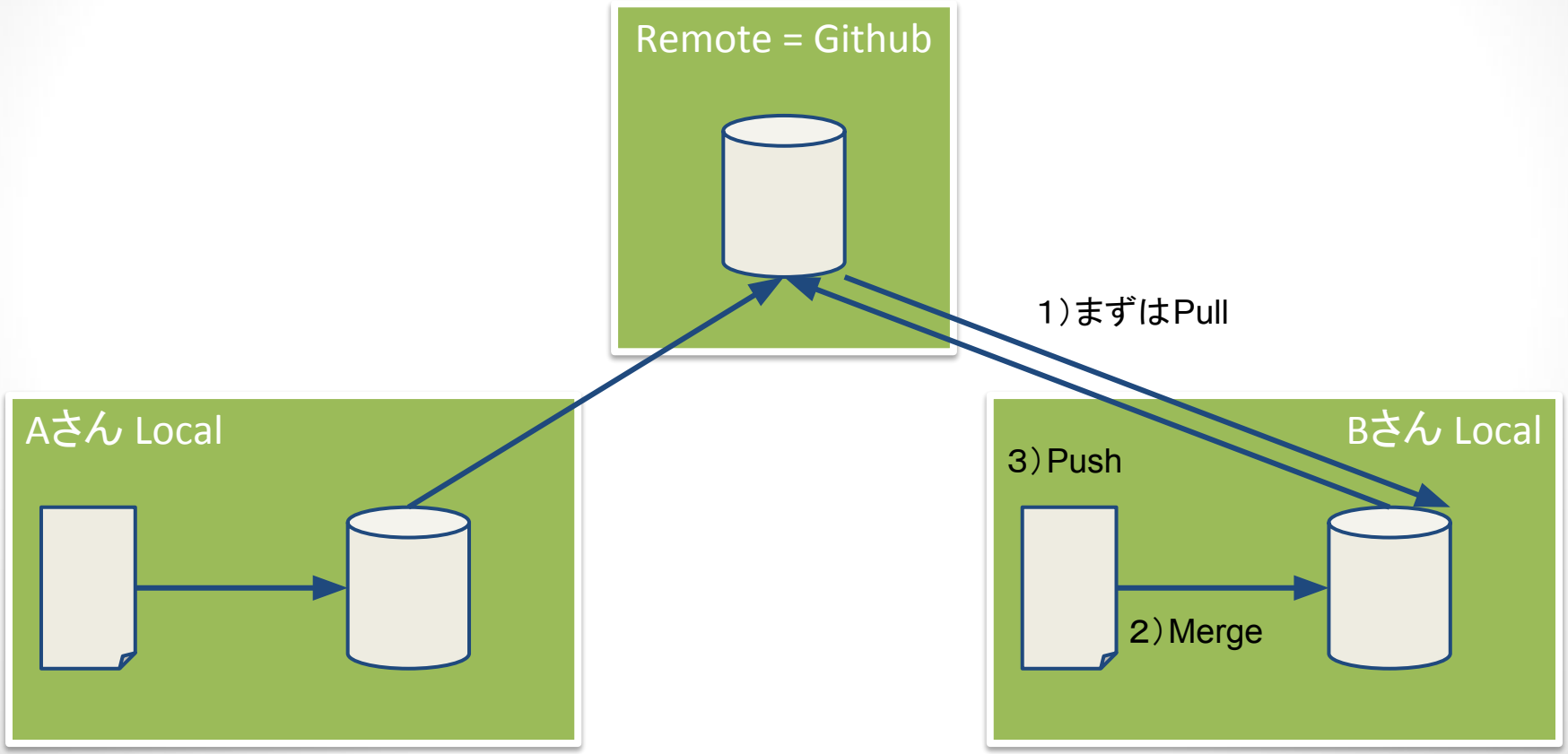
レベル2

Pull, Merge

こういう時...


Remote = Github
上書きされちゃう？
→ Conflictが起きます
Push
Push
Aさん Local
File
C
変更
Bさん Local
File
C
変更

PushでConflictが起きる場合



Gitレベル2

1）Pushしようとすると

```
git push
```

```
To https://github.com/napman/helloworld.git
 ! [rejected]        master -> master (fetch first)
```

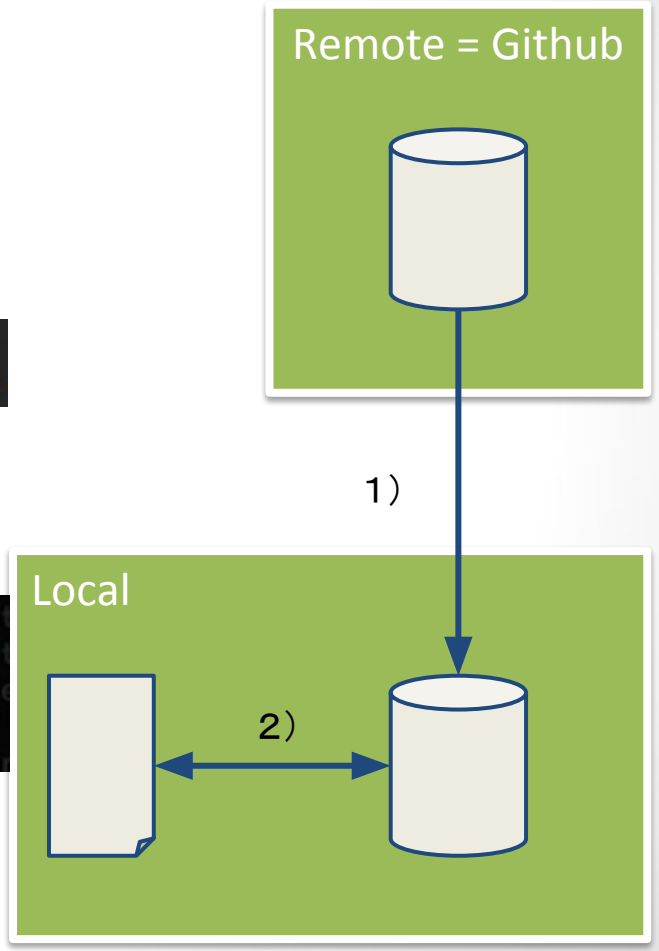
2）Pull（変更の取得）

```
git pull
```

```
From https://github.com/napman/helloworld
   71739aa..16e17f6  master     -> origin/master
Auto-merging hello.py
CONFLICT (content): Merge conflict in hello.py
```

3）Merge

```
git merge
```



Exercise

ブラウザでファイル修正後に*Push*してみる

Git push conflict

Git merge

Gitの問題

でもよく考えるとGit面倒くさくないですかね？

一日何十回gitコマンド実行しないといけないんですか？変更のたびにgit addとかしたくないんですけど。
というか作業効率が著しく下がるのですが。

DiffもMergeも見づらいし、そもそもGitのコマンドオプション覚えたくないんですが。

↓

GUI使えばいいんじゃない？

Demo

Git GUIの勧め

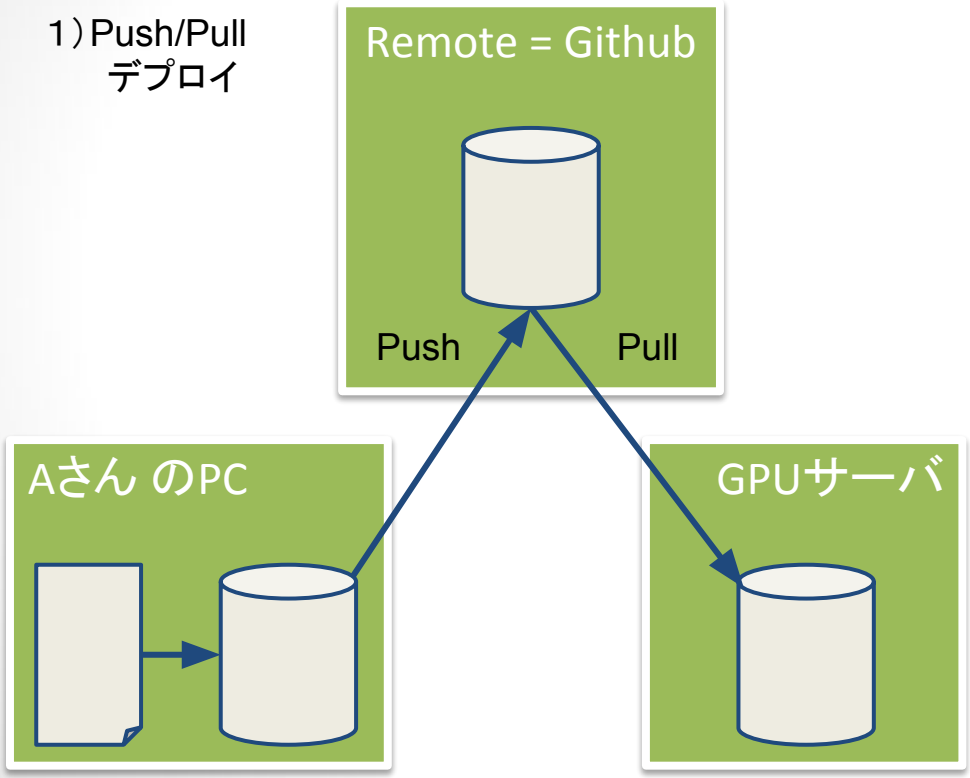
- 不毛なGit Add地獄を回避
- Commit + Push機能でPushの敷居が下がる
  → Remoteがバックアップとして機能する
- Merge機能が健康的
- 強力なHistory機能
- コマンドオプションを覚えなくていい

Git GUIの注意

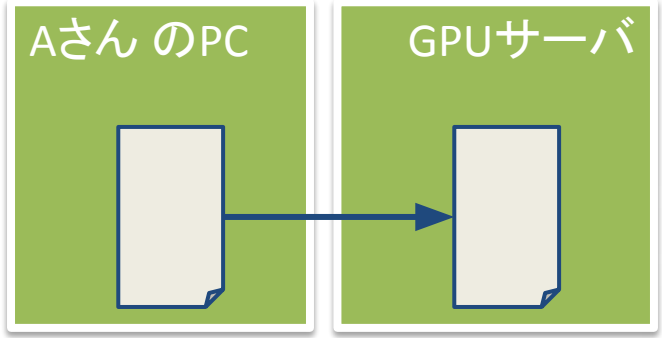
- SSHFSと組み合わせるかRsync、Pullデプロイなど工夫をすれば、GPUサーバとの併用可能
- Git GUIs
  - Github Desktop、SmartGit、SourceTree
  - Git Kraken、GitEye等、別になんでも良い
- SSHFSなどが逆に面倒という人はコマンドラインでいいんじゃないか → 結局は自分が使いやすいソリューションが良い

Gitとデプロイ

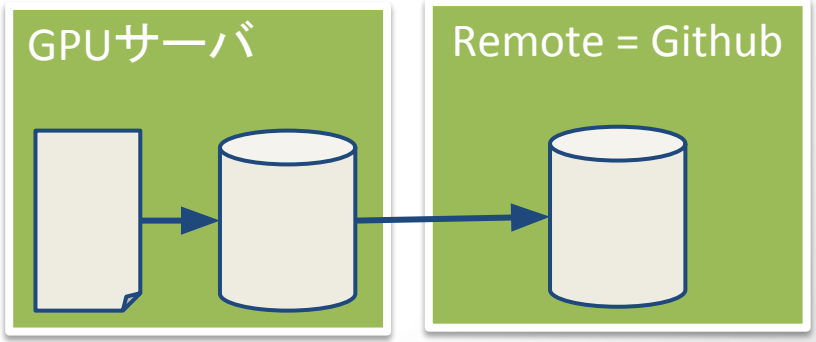
1）Push/Pull
デプロイ



2）Rsyncデプロイ



3）サーバでGit

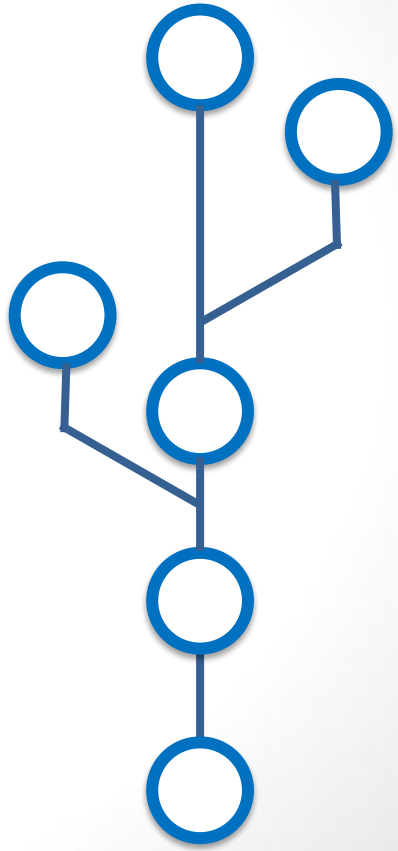


レベル3

ブランチ

ブランチとは

- コミット（変更履歴）の分岐
- こんな時に便利
  - いろんな方向の開発を並行で進めたい
  - 他の人に影響を与えずに作業をしたい
  - 常に**動作可能なコード**を残しておきたい
    （リリースブランチ、master）
  - 大きな変更をするときに、
    いつでも戻せる状態にしておきたい



Git branch

Demo

今日のコンテンツ

- プロジェクトについて
- Git入門
- Gitの作法
- アジャイル開発
- Deep Learningプロジェクト注意
- まとめ
- チームでの議論タイム

Gitの作法

Gitの作法

- 重いデータはリポジトリに入れない
  （Google Drive / Dropbox推奨）
- .gitignoreをちゃんと使う
- README.mdを書く
- README.mdにダラダラ長い文章を書かない。スターティングポイントであることを意識する。
- 他のメンバーやメンターが最小の労力で検証できるように気をつける
- requirements.txtを用意・メンテナンスする

おもてなしの精神

NG集

- 単一ディレクトリにファイルが全部入っている
- コマンドを10個以上叩かないと検証できない
- README.mdがやたらと長い
- 大きいデータセットがリポジトリに入っている
- 大量のログ・ファイルがリポジトリに入っている
- 環境構築のためのスクリプトが無い
- 自分の環境（IDE等）特有のファイルがある

Demo

Gitとワークフロー

Gitとワークフロー

- ワークフローとは
  - Gitの使い方は自由度が高い（高すぎる）
  - ある程度使い方の規約を決めて運用
  - 運用を強制するスクリプトも含める
- 有名なワークフロー
  - Centralized Workflow
  - Feature Branch Workflow
  - Git Flow / Github Flow

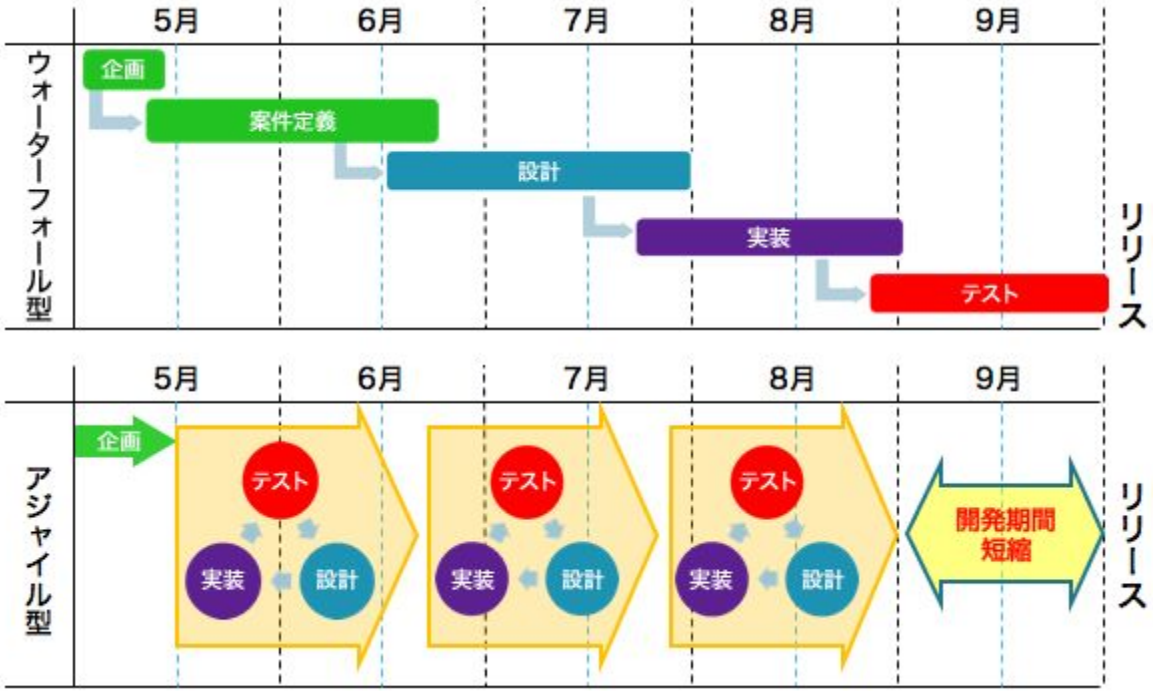
Github Flow

- masterブランチは常時デプロイ可能
- トピックブランチをmasterから作る
- リモートにこまめにPush
- 機能ができたらPull Request＋検証
- 検証完了 → masterへマージ
- すぐにデプロイ → 本番と同期

今日のコンテンツ

- プロジェクトについて
- Git入門
- Gitの作法
- アジャイル開発
- Deep Learningプロジェクト注意
- まとめ
- チームでの議論タイム

ウォーターフォールとアジャイル開発



from www.ves.co.jp

スクラム

スクラムとは

- チームで仕事の進めるための枠組みの一種
- 短い期間の単位で開発を区切り、繰り返す
- 高頻度で優先度を変更する
- 良い点
  - 早めに軌道修正が出来る
  - 短い期間で、最大限の成果をあげる
  - 自立的なチーム作り

スクラム風開発方法

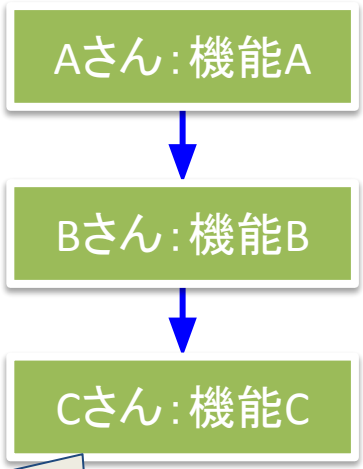
- タスクボード（カンバン型タスクボード）
  - 各自のタスクの可視化
  - 長期的な目標を常に確認
  - タスク・議論のログ
- 定期的なスクラムミーティング
  - Weeklyミーティング
  - Dailyミーティング
  - １５分～３０分程度
- サーヴァントリーダー

Demo

信頼と責任、検証

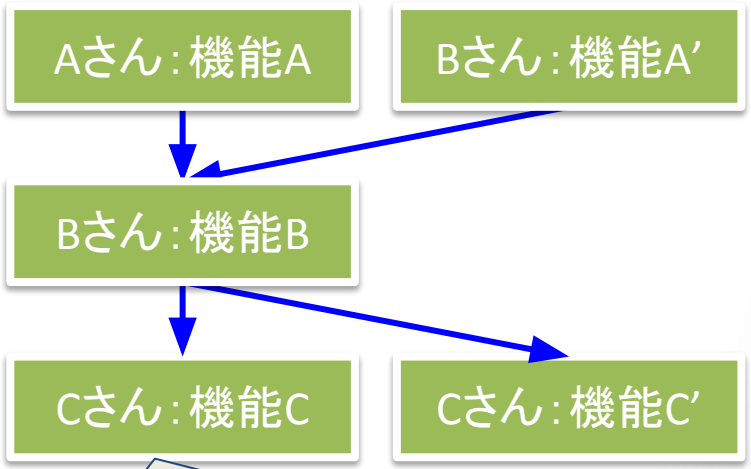
信頼と責任、プランB

失敗するプロジェクト



- 最後に全部結合しよう
- 機能Aはこういうふうに作られるはず
- Aさんを信頼しよう
- あれ？ 思ってたのと違う

丁寧なプロジェクト



- Aさんの案がダメだったときのために A'を用意しておこう
- 複数のモデルを並行で検討しよう
- 先に入力と出力だけは決めておこう

具体的アドバイス

- 最初の数週間で一旦の成果が出るようなものが良い
- 筋が良さそうだと感じたら改良フェーズに入る
- 芳しくない場合（多くの場合はこちら）は、別の方向性の開発も走らせる

今日のコンテンツ

- プロジェクトについて
- Git入門
- Gitの作法
- アジャイル開発
- Deep Learningプロジェクト注意
- まとめ
- チームでの議論タイム

Deep Learning
プロジェクト注意

- DLプロジェクトの特徴
  - とにかく学習・モデル構築に時間がかかる
  - 通常のアプリ開発と違い、分担が難しい
  - データの共有など特殊な事情がある
  - 環境の違いによる影響を受けやすい
- TIPS
  - モデル構築の時間をスケジュールとしてしっかりとっておく
    （講義としては年末・年始あたりが勝負）
  - 複数のモデルを分担して開発
  - すぐに実行できるようにREADME.mdを充実させておく

今日のコンテンツ

- プロジェクトについて
- Git入門
- Gitの作法
- アジャイル開発
- Deep Learningプロジェクト注意
- まとめ
- チームでの議論タイム

まとめ

- DL講座ではGit/Githubが必須
  - 授業の評価としてコミットログを参照
  - 慣れてない人は必ずマスターしておく（宿題）
- 道具に振り回されない、効率が良い方法がベスト
- 自分たちに合ったワークフローを採用する
- プロジェクトの回し方を考える
- お互いに検証する、簡単に検証できるようにする

注意事項

プロジェクト申し込みフォーム

https://goo.gl/pRpYQq

（締め切り：11/06）

今日のコンテンツ

- プロジェクトについて
- Git入門
- Gitの作法
- アジャイル開発
- Deep Learningプロジェクト注意
- まとめ
- チームでの議論タイム